Autotuning Tensor Contraction Computations on GPUs

Axel Rivera, Mary Hall
School of Computing
University of Utah
Salt Lake City, UT
Email: {axelriv,mhall}@cs.utah.edu

Paul D. Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL

Elizabeth Jessup, Thomas Nelson
Department of Computer Science
University of Colorado
Boulder, CO

Boyana Norris
Department of Computer and Information Science
University of Oregon
Eugene, OR

***Abstract*—We describe a framework for generating optimized GPU code for computing tensor contractions, a multidimensional generalization of matrix-matrix multiplication that arises frequently in computational science applications. Typical performance optimization strategies for such computations transform the tensors into sequences of matrix-matrix multiplications to take advantage of an optimized BLAS library, but this approach is not appropriate for small tensors. We instead develop an autotuning strategy that generates CUDA variants from a sequential implementation and identifies the best-performing variant. We compare our generated code with that of OpenACC when offloading the same computation to the GPU. The straightforward OpenACC implementation is as much as 23X slower than our automatically generated code for benchmarks representative of two large-scale tensor contraction computations, Nek5000 and NWChem. However, we show how changes in GPU thread-block decomposition and register placement of data in the OpenACC annotations can achieve comparable performance to our automatically generated code. This result highlights limitations of the OpenACC compiler in targeting GPUs for computations such as tensor contractions with small trip counts and large dimensionality. It also suggests additional optimizations that can overcome these limitations.**

***Keywords*-tensor contraction, spectral element method, code generation, autotuning**

I. Introduction

In this paper, we consider GPU computation of tensor contractions, a multidimensional generalization of matrix-matrix multiplication. Such computations arise frequently in computational science applications. We focus on specific instances from computational fluid dynamics using the spectral element method and electronic structure modeling using coupled cluster theory.

Previous work showed that performance of such tensor contractions on small matrices needed different optimization strategies from the highly tuned BLAS libraries, which are designed for large matrices that do not fit in cache [1]. This prior work examined sequential code running on conventional microprocessors. In this paper, we develop the optimization strategy that targets GPUs. The challenge in targeting GPUs is that parallelization of fine-grained matrix computations must overcome the overhead of copying the data to and from the GPU. In addition, although the large dimensionality and small trip counts of the studied tensor computations provide adequate computation for the GPU, significant tradeoffs remain in mapping the computation to GPU threads and blocks.

We explore many possible code variants for such computations by combining the Orio [2], [3] autotuning and CUDA-CHiLL [4] code transformation tools. We compare the results of our optimization with a similar strategy in OpenACC. The contributions of the work are threefold: a system design for autotuning that integrates two complementary frameworks, its application to tensor contractions, and the performance gains achieved on two representative tensor codes taken from real-world applications. We demonstrate that this approach provides GPU performance far exceeding that provided by OpenACC annotations (up to 23x faster), with a comparable level of programmer effort. We then analyze the performance differences between automatically generated and OpenACC code. By making modest changes to the OpenACC code for thread and block decomposition and placing temporary data in registers, we are able to close most of the performance gap with OpenACC.

The rest of this paper is organized as follows. Section II provides background on tensor contractions. Sections III and IV discuss the Orio and CHiLL tools and describe how we combined them. In Section V, we present experimental results from the Nekbone proxy app [5] and CCSD(T) kernels extracted from NWChem [6], [7]. In Section VI, we place our work in context. In Section VII we summarize our conclusions, and, in Section VIII, we briefly discuss future work.

II. Background

Tensors are a multidimensional generalization of matrices and are a natural way to express many computations arising in scientific computing. The rank of a tensor is the number

of dimensions; a vector is a rank-1 tensor and a matrix is a rank-2 tensor. Two types of tensor computation are particularly common: tensor decompositions, a computation frequently used in data analysis, and tensor contractions, a multidimensional analog of matrix-matrix multiplication used in coupled cluster electronic structure calculations [8], [9], in spectral element discretizations of partial differential equations [10], and as a building block for tensor decompositions. In this paper we focus on tensor contractions—summation along one or more tensor dimensions.

For convenience, we represent tensor contractions using the Einstein summation convention, where whenever the same index appears twice in an expression, once as a superscript and once as a subscript, there is an implied summation over all values of an index. Thus, the vector inner product is represented as $y = u_i v^i$, the matrix-vector product as $y^i = A^i_j x^j$, and the matrix-matrix product as $C^i_k = A^i_j B^j_k$. The contraction of a rank-3 tensor with another rank-3 tensor along one dimension results in a rank-4 tensor

$$C^{ij}_{lm} = A^{ij}_k B^k_{lm} \equiv \sum_k A^{ij}_k B^k_{lm},$$

and the contraction of a rank-3 tensor with another rank-3 tensor along two dimensions results in a rank-2 tensor

$$C^i_l = A^i_{jk} B^{jk}_l \equiv \sum_j \sum_k A^i_{jk} B^{jk}_l.$$

We are interested in computing multidimensional tensor contractions as efficiently as possible. We focus on scenarios featuring computations over thousands of identically sized small tensors (size O(1)–O(10) in each dimension), since these occur naturally in the spectral element method and provide a building block for computations with large tensors in coupled clustered computations. Consider the case of a pth-order spectral element discretization of a PDE on a mesh with N elements. For each mesh element, one must compute tensor contractions of the form

$$V_{ij} = A^l_j B^k_i U_{kl} \equiv \sum_{k=0}^{p} \sum_{l=0}^{p} A^l_j B^k_i U_{kl}$$

in two dimensions or

$$V_{ijk} = A^l_k B^m_j C^n_i U_{lmn} \equiv \sum_{l=0}^{p} \sum_{m=0}^{p} \sum_{n=0}^{p} A^l_k B^m_j C^n_i U_{lmn}$$

in three dimensions. A naive implementation of the two-dimensional contraction requires O(p^4) operations (O(p^2) for each of the p^2 members of V_{ij}). However, this approach ignores redundant subcomputations across columns and rows of V. One can instead compute $W^l_i = B^k_i U_{kl}$ followed by $V_{ij} = A^l_j W_{il}$ at a cost of O(p^3) operations. A similar reorganization of the three-dimensional computation reduces the cost from O(p^6) operations to O(p^4) operations. Tools such as the Tensor Contraction Engine (TCE) [8], [11], libtensor [12] seek to reorganize tensor contractions in this fashion in order to minimize the number of floating point operations [11]. In this paper, we assume that such reorganizations have already been performed, manually or by using a tool such as TCE. Instead, we focus on how to use loop transformations to expose the parallelism and data locality required for the efficient computation of tensor contractions on GPUs.

III. Code Generation

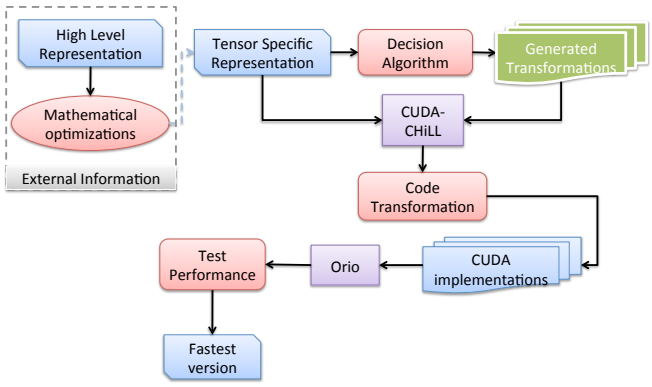

Figure 1. Representation of the framework for Tensor Contraction code generation

To generate many possible variants of a tensor computation and to identify the best-performing implementation, we combined two tools: Composing High-Level Loop transformations (CHiLL) [4] and Orio [2], [3]. Figure 1 shows how these tools work in collaboration. The framework presented in this figure depends on external information that is an expansion to this work. The user inputs a high-level representation that resembles the mathematical tensor notation, and the external tool applies different optimizations for reducing the number of operations among others. The output from the external information is a tensor- specialized representation that is used through this system to generate high-performance code. Section V presents an example of this representation.

The tensor representation from the external tool is used to generate a basic C++ implementation. At a high level, the tool creates a *for* loop for each different index listed in the operation and then generates the statement using the equation. The C++ output from the external tool is used with a decision algorithm to generate a series of transformations. This algorithm is explained in more detail in Section IV.

The framework send the transformations generated to CHiLL, which converts the C++ code into different implementations. CHiLL is a source-to-source compiler with an underlying polyhedral transformation and code generation framework. It uses a transformation recipe that specifies a series of transformations and applies them over a loop nest. CHiLL has an extension CUDA-CHiLL [13] whereby users can generate CUDA code from the sequential version. The

resultant code can replace the original loop nest to achieve higher performance on the CPU or to perform the work on a GPU. The loop transformations expressed in the CHiLL recipe can be applied to the GPU code, too.

The different versions generated by CUDA-CHiLL are then analyzed using Orio. Orio is a tool that permits users to annotate their code for specifying different parameterized code transformations as well as potential values or ranges for the code transformation parameters. Orio creates a skeleton code to test the empirical performance, executes a search over the parameter space, and determines which parameter values produce the best version. This search can be specified by the user, and it can invoke different techniques such as exhaustive, Nelder-Mead simplex, and simulated annealing. Orio also supports code transformations, but for this work, we rely on CHiLL for all code transformations and use Orio to search the performance parameter space. Orio was developed in Python and can be expanded by adding modules; we have added CHiLL as an external module to Orio, which permits Orio to read the annotations and transform them into CHiLL scripts.

The CHiLL scripts generate CUDA code with all the necessary information (data allocation, data copies, kernels, and kernel calls) and replace the original function with an autotuned version. Orio uses the skeleton code to measure the execution time for each generated kernel, using a user-defined number of repetitions. Then, Orio computes the average of these repetitions for each version and selects the fastest average implementation as the final output.

IV. Autotuning/Decision Algorithm

The goal of the framework is to automatically find a fast implementation oriented to small tensor contraction problems on GPUs. To achieve this goal, we implemented a decision algorithm to find those transformations that improve the performance as well as to discover an efficient thread and block decomposition. Our algorithm is a simplified version of Khan et al.'s algorithm [14], [15]. The algorithm finds the thread and block decomposition and the data that need to be placed in different memory levels. Then it applies different transformations at the thread level to improve the performance. Using autotuning, the algorithm finds the parameters for the transformations that achieve the best performance.

The decision algorithm starts by performing a data dependency analysis. For the purpose of tensor contractions, this analysis verifies which indices are present in the right-hand side but not in the left-hand side of the operation. These values indicate that they carry a loop dependence, and the other iterations can be performed in parallel. The dependence analysis also provides information about which of the input tensors achieve contiguous access to the data available in memory. The tensor where loop indices are represented from innermost to outermost achieves linear access to the data. This tensor is named as the *contiguous tensor*.

The decision algorithm uses the dependence analysis and autotuning to generate the thread and block decomposition. It sets the innermost loop that doesn't carry a dependence in the contiguous tensor as $Thread_X$ on the GPU. $Thread_Y$, $Block_X$, and $Block_Y$ are generated by using a variation after selecting the potential loops by the following rules:

- Select parallel indices from the *contiguous tensor* from innermost to outermost loops.
- If the *contiguous tensor* has fewer than four parallel loops, then start selecting parallel indices from the *non-contiguous tensor* from outer to inner.

The algorithm target is selecting the loops that access contiguous data. Contiguous access is important because it permits the use of *memory coalescing*, a feature available on GPUs that allows the transfer of contiguous data from a higher memory level to registers in a single transfer.

After selecting the loops that are going to be performed at the threads and blocks level, the algorithm permutes those loops that are left inside the kernel level if the data is column-major. This step is important in order to reduce the cache misses that can be caused by the noncontiguous tensor. Also, the decision algorithm moves to the registers the data that can be reused in the loop that carries a dependence, in order to reduce the communication with the global memory.

The decision algorithm will test different variations of $Thread_Y$, $Block_X$, and $BlockY$ with the optimization strategies presented. Then, using an average from all runs, it selects the fastest decomposition and unroll the loop that carries the dependencies, in order to reduce the control flow inside the kernel. The unroll parameters are those factors that evenly divide the iteration space. These parameters prevents the introduction of conditionals. The framework automatically tests all the unroll parameters to decide on the fastest implementation.

V. Experimental Results

We evaluated the performance of the code generated by the integrated Orio-CHiLL system for excerpts from two tensor contraction applications: Nekbone and NWChem.

Nekbone is a 3-dimensional spectral element proxy application derived from Nek5000 [16], [17]. It performs a conjugate gradient loop that operates over a sequence of tensor contractions recast as matrix multiplications, which comprises 60% of the sequential execution time. Each iteration computes the following for each element:

- $Ur_{ijk} = D_{il}U_{ljk}$: recast as a $p \times p$ matrix multiplied by a $p \times p^2$ matrix
- $Us_{ijk} = D_{jl}U_{ilk}$: recast as p matrix-matrix multiplies with $p \times p$ matrices
- $Ut_{ijk} = D_{kl}U_{ijl}$: recast as a $p^2 \times p$ matrix multiplied by a $p \times p$ matrix

Nekbone was tested with three problem sizes: $8 \times 8 \times 8$, $10 \times 10 \times 10$, and $12 \times 12 \times 12$. The small sizes are caused by the order of the discretization polynomial; as it increases, the time required to converge also increases.

NWChem is a software package for quantum chemistry and molecular dynamics simulations [6]. We optimized kernels [7] extracted from the CCSD(T) (coupled cluster theory with full-treatment singles and doubles, and triples estimated by using perturbation theory) computations of NWChem. The main kernel is divided into three sets where each set has nine functions that explore the tensor contraction in different points of the data. The first (*sd_t_s1*) set performs a series of operations over two objects with two and four dimensions. The second (*sd_t_d1*) and third (*sd_t_d2*) sets work with two objects each having four dimensions. All three cases store the results into a six-dimensional object. We tested our implementation when the dimension sizes are 10, 12, and 16. These sizes represents chunks of a larger tensor contraction that is computed in a distributed system.

A. Methodology

To evaluate the performance of the codes generated and optimized by our new system, we extracted the functions corresponding to the computations described above. We implemented the tensor representation of these functions and passed them to the framework for generating the CUDA implementation. Listing 1 presents the input representation for the tensor contraction found in Nekbone. At a high-level, the tensor representation specifies how memory is accessed and defines values to use across the file. Then, using the tensors specified in the *variables* block, it presents the tensor contraction with the corresponding indices. The framework can also handle the cases where dimensions are combined to improve the performance as explained in Section II.

```
local_grad3
memory: column
access: linearize
defines:
  lx = 10
  ly = 10
  lz = 10
  nelt = 100
variables:
  u:(nelt,lx,ly,lz)
  D:(lx,ly)
  Dt:(lx,ly)
  ur:(nelt,lx*ly,lz)
  us:(nelt,lx,ly,lz)
  ut:(nelt,lx,ly*lz)
operation:
  ur:(e,j,i) += D:(k,i)*u:(e,j,k)
  us:(e,j,i,k) += u:(e,j,m,k)*Dt:(i,m)
  ut:(e,j,i) += u:(e,k,i)*Dt:(j,k)
```

Listing 1. Tensor representation for the computation performed in Nekbone

The tensor specific representation is used to generate the different decisions using the algorithm presented in Section IV. It computes the average of 100 runs to select the fastest thread and block decomposition. After the grid is created, the skeleton code provided by Orio tests each unroll factor 100 times and computes the average execution time. The fastest implementation will replace the original kernel in the benchmarks. We compared the performance of the generated code from the framework with that of OpenACC, which is a pragma interface for specifying which computations should be offloaded to a GPU.

We performed a set of experiments on these codes on an Intel i7 930 with 2.8 GHz CPU, 4 GB of RAM, and Ubuntu 14.04 64-bits. The GPUs used were NVIDIA TESLA C2050 (Fermi) and TESLA K20 (Kepler). For the OpenACC results, we used the Portland Group compiler (PGI) version 14.3. The CUDA code was compiled by using the nvcc compiler for CUDA 5.5.

B. Results

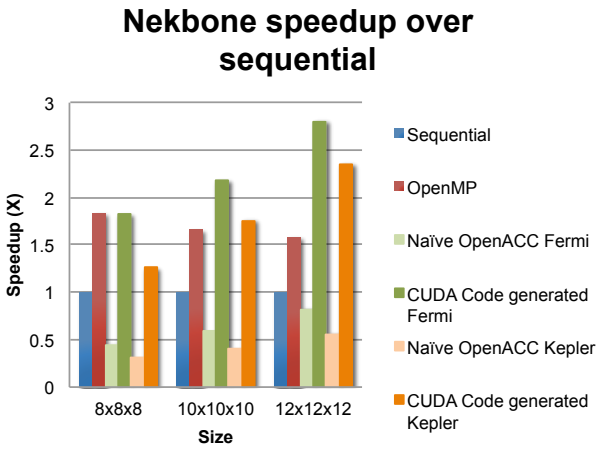

Figure 2. Speedups achieved over sequential implementation in Nekbone

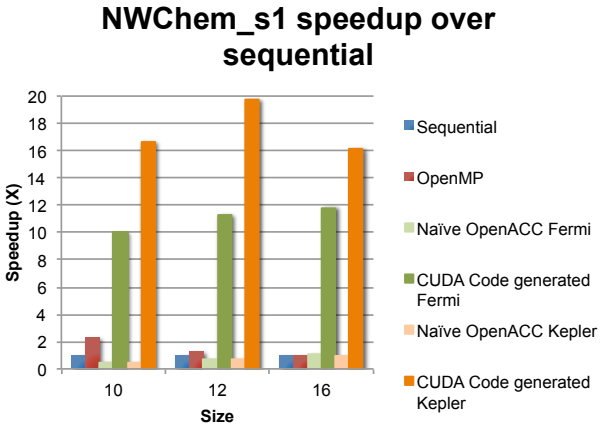

Figure 3. Speedups achieved over sequential implementation in NWChem sd_t_s1

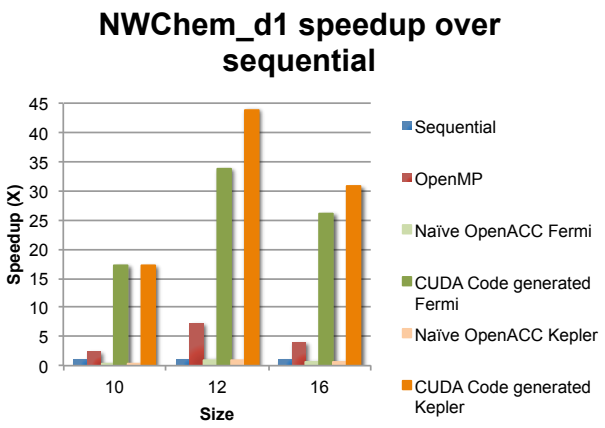

Figure 4. Speedups achieved over sequential implementation in NWChem sd_t_d1

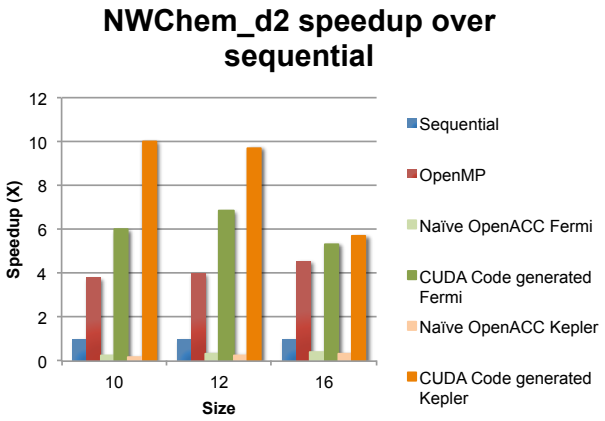

Figure 5. Speedups achieved over sequential implementation in NWChem sd_t_d2

Results from the experiments related to Nekbone are shown in Figures 2. Figures 3, 4, and 5 corresponds to the three different kernels of NWChem. These results are the measurements of four versions: sequential (blue bars), OpenMP with 4 threads (red), OpenACC (light color), and the code generated from the system (dark color) in Fermi (green gradients) and Kepler (orange gradients). In the OpenACC version we allowed the compiler to decide which optimizations should be applied as well as the thread and block decomposition. This version is henceforth referred to as naïve OpenACC.

In both versions that use GPUs, the instructions related to data copies and memory allocation were moved outside of the main computation. The Nekbone benchmarks that make use of GPUs were modified to use asynchronous copies and CUDA streams for launching multiple kernels at the same time. This was possible because the matrix products *Ur*, *Us* and *Ut* are independent. We couldn't use these optimizations for NWChem because the kernels carry a data dependency. The results include the overhead of data copy and memory allocation as part of the execution time.

Speedup achieved by code generated in Nekbone				
Size	GPU	Sequential	OpenMP	OpenACC
$8 \times 8 \times 8$	Fermi	**1.82x**	0.99x	0.45x
	Kepler	**1.26x**	0.68x	0.30x
$10 \times 10 \times 10$	Fermi	**2.17x**	1.39x	0.59x
	Kepler	**1.75x**	1.06x	0.40x
$12 \times 12 \times 12$	Fermi	**2.80x**	1.77x	0.82x
	Kepler	**2.35x**	1.49x	0.56x

Table I
SPEEDUPS ACHIEVED BY THE CODE GENERATED IN THE FRAMEWORK FOR NEKBONE

Speedup achieved by code generated in NWChem					
Kernel	Size	GPU	Sequential	OpenMP	OpenACC
sd_t_s1	10	Fermi	**10x**	4.35x	0.48x
		Kepler	**16.67x**	7.26x	0.45x
	12	Fermi	**11.28x**	9x	0.68x
		Kepler	**19.75x**	15.75x	0.70x
	16	Fermi	**11.85x**	11.8x	1.06x
		Kepler	**16.16x**	16.09x	0.94x
sd_t_d1	10	Fermi	**17.23x**	7.13x	0.36x
		Kepler	**17.23x**	7.13x	0.26x
	12	Fermi	**33.74x**	4.67x	1.07x
		Kepler	**44x**	6.08x	0.82x
	16	Fermi	**26.09x**	6.54x	0.71x
		Kepler	**30.78x**	7.72x	0.67x
sd_t_d2	10	Fermi	**6x**	1.58x	0.25x
		Kepler	**10x**	2.63x	0.19x
	12	Fermi	**6.85x**	1.72x	0.36x
		Kepler	**9.72x**	2.43x	0.20x
	16	Fermi	**5.37x**	1.18x	0.39x
		Kepler	**5.71x**	1.25x	0.25x

Table II
SPEEDUPS ACHIEVED BY THE CODE GENERATED IN IN THE FRAMEWORK FOR NWCHEM

Table I presents the speedups achieved in Nekbone by the code generated from the framework. Results show that the performance improved by up to 2.80x over the sequential execution. Compared to OpenMP, we achieved up to 1.77x speedup. Our results indicate that the performance improvement scales with the data size. The improvement is due to the GPUs' ability to process more data in parallel.

The results for NWChem are shown in Table II. We observed that *sd_t_d1* achieved up to 19.75x speedup over the sequential and 15.75x over the OpenMP implementations. For *sd_t_d1*, the autotuned code increased the performance up to a factor of 44x and 7.72x over the sequential and OpenMP implementations. The results for *sd_t_d2* present speedups of 10x and 2.63x over other versions. The massive speedup achieved in *sd_t_d1* is caused by the data access patterns in the loop nests. NWChem uses the same data across all three kernels, but each kernel touches it differently. The data requested by the iterations of *sd_t_d1* are located closer to each other in comparison with *sd_t_s1* and *sd_t_d2*.

This enables reuse of more data already available in cache. The same performance behavior can be seen in OpenMP and OpenACC.

C. Comparison with OpenACC

For all the benchmarks, the naïve OpenACC version was substantially slower than the sequential implementation of the code. The observed speedup for Nekbone ranges from 0.30x to 0.82x compared to the sequential time; in the case of NWChem, it ranges from 0.19x to 1.07x. We used the NVIDIA Visual Profiler to study the source of the slow-downs in the naïve OpenACC version. Listing 2 presents the naïve implementation of the NWChem *sd_t_d1_1* function. The results from profiling this code indicate that the thread and block decomposition generated by the compiler do not use the GPU efficiently. The decomposition generated by the compiler for the case where each dimension is of size 16 is as follows.

	Block	Thread
X	32	1
Y	1	16

The profiler revealed that this decomposition provides only 50% of the warp execution efficiency. In other words, the code does not take advantage of the scheduling unit of the GPU. To optimize the performance of OpenACC, we decided to implement the thread and block decomposition found by the decision algorithm and also explicitly move the reduction variables to registers.

```
#pragma acc kernels loop present(t2[0:tile4],
 v2[0:tile4], t3[0:tile6]) {
#pragma acc loop independent
 for (int p4=0; p4<p4u; p4++){
#pragma acc loop independent
 for (int p5=0; p5<p5u; p5++){
#pragma acc loop independent
 for (int p6=0; p6<p6u; p6++){
#pragma acc loop independent
 for (int h1=0; h1<h1u; h1++){
#pragma acc loop independent
 for (int h2=0; h2<h2u; h2++){
#pragma acc loop independent
 for (int h3=0; h3<h3u; h3++){
 for (int h7=0; h7<h7u; h7++){
  t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u*
    p4)))))] -= t2[h7+h7u*(p4+p4u*(p5+p5u*
     h1)))] * v2[h3+h3u*(h2+h2u*(p6+p6u*h7))];
}
```

Listing 2. Naïve OpenACC implementation of sd_t_d1_1

Listing 3 presents the OpenACC implementation after indicating the following thread and block decomposition.

	Block	Thread
X	16	16
Y	16	16

This optimization allowed OpenACC to increase the memory bandwidth related to the communication between L1 and L2 cache. Also, moving the reduction variable to the register level, as presented in Listing 4, improved the bandwidth related to device memory. Figure 6 presents the difference in memory bandwidth between the naïve implementation and after applying the optimizations. The data reads (red bars) associated with L1/L2 cache increased from 6.593 GB/s to 258.672 GB/s, while the writes (blue) gain a boost from 5.023 GB/s to 13.614 GB/s. In the results for the device memory bandwidth, the data reads (purple) grow from 0.408 GB/s to 17.632 GB/s and writes (green) from 6.2 GB/s to 17.001 GB/s. These optimizations improved the reuse of data available in cache memory, which reduces the communication with the global memory. The profiler also showed that the new implementation uses 100% of the warp execution efficiency. We specified by hand the information copied to registers in the third implementation since the *private* clause in OpenACC did not work properly. We expect that this problem will be resolved in future OpenACC compiler implementations.

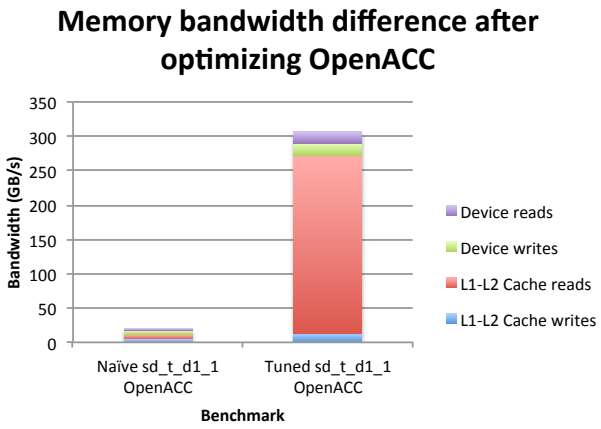

Figure 6. Memory bandwidth performance related to naïve OpenACC implementation and tuned OpenACC

```
#pragma acc kernels loop present(t2[0:tile4],
 v2[0:tile4], t3[0:tile6]) {
#pragma acc loop independent
 for (int p4=0; p4<p4u; p4++){
#pragma acc loop independent
 for (int p5=0; p5<p5u; p5++){
#pragma acc loop independent gang(16)
 for (int p6=0; p6<p6u; p6++){
#pragma acc loop independent gang(16)
 for (int h1=0; h1<h1u; h1++){
#pragma acc loop independent vector(16)
 for (int h2=0; h2<h2u; h2++){
#pragma acc loop independent vector(16)
 for (int h3=0; h3<h3u; h3++){
 for (int h7=0; h7<h7u; h7++){
  t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u*
    p4)))))] -= t2[h7+h7u*(p4+p4u*(p5+p5u*
     h1)))] * v2[h3+h3u*(h2+h2u*(p6+p6u*h7))];
}
```

Listing 3. OpenACC implementation of sd_t_d1_1 with thread and block decomposition specified

Speedup achieved by tuning OpenACC				
Benchmark	GPU	Naïve	New Grid	Registers
Nekbone	Fermi	0.82x	1.85x	1.87x
	Kepler	0.56x	1.57x	1.61x
sd_t_s1	Fermi	1.06x	12.93x	
	Kepler	0.94x	17.77x	
sd_t_d1	Fermi	0.71x	12.07x	21.99x
	Kepler	0.67x	16.91x	30.78x
sd_t_d2	Fermi	0.39x	4.5x	4.60x
	Kepler	0.25x	6.36x	6.65x

Table III
SPEEDUP ACHIEVED BY TUNING OPENACC WITH DIFFERENT OPTIMIZATIONS

```
#pragma acc kernels loop present(t2[0:tile4],
 v2[0:tile4], t3[0:tile6]) {
 double vard1_1;
#pragma acc loop independent
 for (int p4=0; p4<p4u; p4++){
#pragma acc loop independent
 for (int p5=0; p5<p5u; p5++){
#pragma acc loop independent gang(16)
 for (int p6=0; p6<p6u; p6++){
#pragma acc loop independent gang(16)
 for (int h1=0; h1<h1u; h1++){
#pragma acc loop independent vector(16)
 for (int h2=0; h2<h2u; h2++){
#pragma acc loop independent vector(16)
 for (int h3=0; h3<h3u; h3++){
  vard1_1 = t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+
            p6u*(p5+p5u*p4))))];
 for (int h7=0; h7<h7u; h7++){
   vard1_1 -= t2[h7+h7u*(p4+p4u*(p5+p5u*
     h1))] * v2[h3+h3u*(h2+h2u*(p6+p6u*h7))];
 }
  t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u
     *p4))))] = vard1_1;
}
```

Listing 4. OpenACC implementation of sd_t_d1_1 with thread and block decomposition specified and data copied to registers

Table III presents the speedups achieved by the optimized OpenACC implementations compared with the sequential implementation. When the thread and block decomposition identified by the decision algorithm was included in the OpenaCC implementation of Nekbone with size $12\times12\times12$, the performance increased to 1.85x over the sequential implementation. After moving the data to the registers level, we saw a modest increase of a factor 1.87x faster. For *sd_t_s1* with size 16, providing the thread and block decomposition yields speedups up to 17.77x over the sequential implementation. This kernel doesn't use the registers level optimization since there is no reduction variable. The *sd_t_d1* kernel performance increased to 16.91x with the explicitly specified decomposition, and moving the data to registers yielded a further improvement to 30.78x over sequential. NWChem *sd_t_d2* achieved 6.36x speedup over the sequential implementation after using the thread and block decomposition from the decision algorithm, and it improved to 6.65x after copying the data to the registers.

Figure 7 presents a comparison between the tuned OpenACC implementations and the codes generated by our

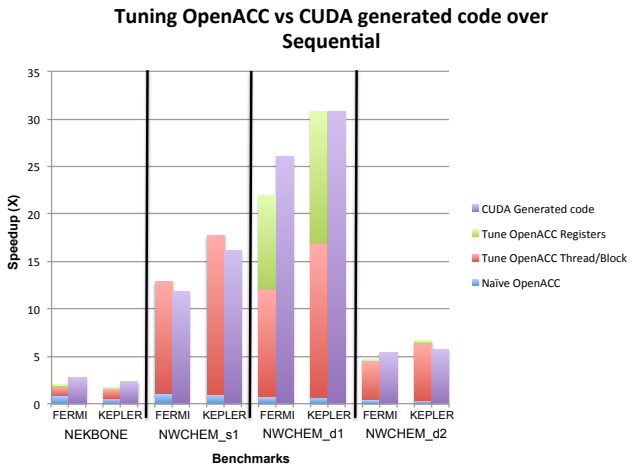

Figure 7. Performance comparison between tuning OpenACC and the code generated from the framework

system. In some cases, tuning for OpenACC outperforms the CUDA code created by the framework. The optimized OpenACC version of NWChem *sd_t_s1*, with size 16, performed 1.09x faster on Fermi and 1.1x on Kepler. The optimized OpenACC implementation of *sd_t_d1* on Kepler achieved the same performance as the CUDA code generated by the autotuning framework. The results for *sd_t_d2* are mixed; on FERMI, the generated CUDA code outperforms the optimized OpenACC implementation, while on Kepler the optimized OpenACC performed 1.16x faster than the fastest CUDA implementation. These results present that the optimizations from the OpenACC compiler, in conjunction with the decisions found by the decision algorithm, is possible to achieve equal or higher performance than developing high-performance CUDA code. We observed a potential of using OpenACC as a target for autotuning instead of CUDA.

VI. RELATED WORK

Many tools and libraries have been developed for optimizing tensor contraction computations. Among these are the tensor contraction engine (TCE), the super instruction assembly language (SIAL) [18], and Cyclops [9]. These efforts have focused primarily on contractions involving very large tensors, possibly distributed across a large parallel computer. Performance optimization focuses on reducing the number of operations performed, by exploiting symmetry and redundant subexpressions. Often, tensors are transposed so that a high-performance matrix-matrix multiplication can be used. Our focus is on the small tensors that arise in the spectral element method and can be used as a building block for computations involving large tensors. Thus, different optimization priorities apply.

Several efforts have sought to optimize the performance of the Nekbone proxy application on GPUs. The CESAR codesign center reports tensor contraction performance of

100–200 GFlops on a Fermi GTX 590 GPU for tensors of size $8 \times 8 \times 8$ to $12 \times 12 \times 12$ using hand-coded OpenCL kernels [19]. The CRESTA project ported Nekbone to a multi-GPU system and reported a speedup of 1.59x using 512 Nvidia Kepler K20x GPUs versus a CPU-only implementation (512 nodes with 8192 cores) [20]. Although direct comparisons are difficult, our speedup of 1.3x versus OpenMP is encouraging, especially since our results include the time to transfer data back and forth between CPU and device memory.

VII. Conclusion

This paper presents a new system integrating CHiLL and Orio that enables automated acceleration of tensor contraction problems for small sizes. The system uses a tensor specific representation to generate high-performance CUDA code. It depends on a decision algorithm that automatically finds the thread and block decomposition, as well as the optimizations that are needed to achieve performance gains over sequential and multicore execution on GPUs. Results from the experiments show speedups that range from 1.26x to 44x compared with the sequential version and 1.06x to 16.09x compared with OpenMP. They also show that OpenACC benefits from the guidance provided by the decision algorithm to achieve better performance. After specifying the grid decomposition and moving some of the data to registers, OpenACC performed up to 53.87x faster than the compiler-parallelized version. For the tensor operations we considered, the compiler-based acceleration is generally poor, whereas both our optimized OpenACC-based parallelization and the direct CUDA code generation and autotuning result in significant performance gains compared with the sequential and OpenMP versions.

VIII. Future work

We are currently incorporating into our tool chain an autotuning decision algorithm based on the Transformation Strategy Generator in [13] that will restrict the search space to transformation parameter values that are likely to be profitable. We will also incorporate autotuning search algorithms [21] to further reduce the number of code variants explored.

Our goal is to support the specification of a sequence of tensor computations in a tensor DSL and automatically generate high-performance, thread-parallel implementations in a fully automated process. This work builds on the current Orio-CHiLL integration as well as on earlier work on the Build-to-Order BLAS [22], [23] for matrix computations. In contrast to TCE and similar tools, our focus is on maximizing the performance of collections of tensor contractions involving small tensors, making specialization and empirical autotuning critical ingredients in the process.

Acknowledgment

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357. Utah researchers were partially supported by DOE award DE-SC0008682 and National Science Foundation award CCF-1018881.

References

[1] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, "Speeding up Nek5000 with autotuning and specialization," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10, 2010, pp. 253–262.

[2] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161004

[3] B. Norris, A. Hartono, and W. Gropp, "Annotations for productivity and performance portability," in *Petascale Computing: Algorithms and Applications*, ser. Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007, pp. 443–462, preprint ANL/MCS-P1392-0107. [Online]. Available: http://www.mcs.anl.gov/uploads/cels/papers/P1392.pdf

[4] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 50–64. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13374-9_4

[5] "Proxy apps for thermal hydraulics," 2014. [Online]. Available: https://cesar.mcs.anl.gov/content/software/thermal_hydraulics

[6] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477 – 1489, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465510001438

[7] "NWChem TCE CCSD(T) loop-driven kernels," https://github.com/jeffhammond/nwchem-tce-triples-kernels.

[8] G. Baumgartner, D. E. Bernholdt, D. Cociorva, C.-C. Lam, J. Ramanujam, R. Harrison, M. Noolijen, and P. Sadayappan, "A performance optimization framework for compilation of tensor contraction expressions into parallel programs," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, ser. IPDPS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 33–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=645610.661695

[9] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS '13)*, May 2013, pp. 813–824.

[10] M. Deville, P. Fischer, and E. Mund, *High-Order Methods for Incompressible Fluid Flow*. Cambridge, 2002.

[11] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. Bernholdt, S. Hirata, C.-C. Lam, R. Pitzer, J. Ramanujam, and P. Sadayappan, "Automated operation minimization of tensor contraction expressions in electronic structure calculations," in *Computational Science ICCS 2005*, ser. Lecture Notes in Computer Science, V. Sunderam, G. van Albada, P. Sloot, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2005, vol. 3514, pp. 155–164.

[12] E. Epifanovsky, M. Wormit, T. Ku, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. I. Krylov, "New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations," *Journal of Computational Chemistry*, vol. 34, no. 26, pp. 2293–2309, 2013. [Online]. Available: http://dx.doi.org/10.1002/jcc.23377

[13] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. n. Chame, "A script-based autotuning compiler system to generate high-performance CUDA code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013.

[14] M. M. Z. M. Khan, "Autotuning, code generation and optimizing compiler technology for GPUs," Ph.D. dissertation, University of Southern California, 2012.

[15] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400690

[16] H. M. Tufo and P. F. Fischer, "Terascale spectral element algorithms and implementations," in *ACM/IEEE conference on Supercomputing*, Portland, OR, USA, 1999.

[17] J. W. L. Paul F. Fischer and S. G. Kerkemeier, "Nek5000 Web page," 2008, http://nek5000.mcs.anl.gov.

[18] B. A. Sanders, R. Bartlett, E. Deumens, V. Lotrich, and M. Ponton, "A block-oriented language and runtime system for tensor algebra with very large arrays," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.3

[19] The CESAR Team, "The CESAR codesign center: Early results," 2012. [Online]. Available: https://cesar.mcs.anl.gov/content/cesar-codesign-center-early-results

[20] "Large-scale fluid dynamics simulations—towards a virtual wind tunnel," 2014. [Online]. Available: ftp://www.mech.kth.se/pub/adam/CRESTA/Case_study/Nek5000_case_study.pdf

[21] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can search algorithms save large-scale automatic performance tuning?" *Procedia Computer Science*, vol. 4, pp. 2136–2145, 2011, proceedings of the International Conference on Computational Science, ICCS, 2011.

[22] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek, "Automating the generation of composed linear algebra kernels," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 59:1–59:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654119

[23] T. Nelson, G. Belter, J. G. Siek, E. Jessup, and B. Norris, "Reliable generation of high-performance matrix algebra," *ACM Transactions on Mathematical Software*, 2014, to appear.